FuzzyData: A Scalable Workload Generator for Testing Dataframe Workflow Systems

Mohammed Suhail Rehman
University of Chicago
Chicago, USA
suhail@uchicago.com

Aaron J. Elmore
University of Chicago
Chicago, USA
aelmore@cs.uchicago.edu

ABSTRACT

Dataframes have become a popular means to represent, transform and analyze data. This approach has gained traction and a large user base for data science practitioners - resulting in a new wave of systems that implement a dataframe API but allow for performance, efficiency, and distributed/parallel extensions to systems such as R and `pandas`. However, unlike relational databases and NoSQL systems with a variety of benchmarking, testing, and workload generation suites, there is an acute lack of similar tools for dataframe-based systems. This paper presents FuzzyData, a first step in providing an extensible workflow generation system that targets dataframe-based APIs. We present an abstract data processing workflow model, random table and workflow generators, and three clients implemented using our model. Using FuzzyData, we can encode a real-world workflow or randomly generate workflows using various parameters. These workflows can be scaled and replayed on multiple systems to provide stress testing, performance evaluation, and a breakdown of performance bottlenecks present on popular dataframe systems.

CCS CONCEPTS

• **Information systems → Database performance evaluation**.

KEYWORDS

dataframe systems, benchmark, workflow generation

ACM Reference Format:
Mohammed Suhail Rehman and Aaron J. Elmore. 2022. FuzzyData: A Scalable Workload Generator for Testing Dataframe Workflow Systems. In *9th International Workshop of Testing Database Systems (DBTest'22), June 17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3531348.3532178

1 INTRODUCTION

The rise in popularity of data science and machine learning has catapulted dataframe-based tools such as R [7] and the `pandas` [20] library for Python to the forefront of the data science practice, enabling large and small organizations to extract insight from data quickly. These tools allow for ad-hoc ingestion and transformation of small to moderate amounts of data, with the flexibility of integrating arbitrary code or machine learning workflows to the data analysis workflows.

The dataframe model has thus become a popular programmatic interface to encode data manipulation and transformation operations. Typical dataframe APIs enable both spreadsheet-style manipulation and relational-style operations and on tabular data, allowing for data cleaning and wrangling during the ETL process while retaining most of the powerful querying semantics for analytical and visualization tasks. With the rise in popularity of this model, expectations of such systems have grown, resulting in the growing demand for systems that support these APIs on massive tabular datasets that may not fit a typical desktop or laptop computers' memory. Projects such as dask [17], ray [13], modin [14], SparkDataFrames [1], either provide interfaces to python scientific tools such as NumPy and pandas or implement their own version of the dataframe API to work on parallel/distributed environments with larger datasets.

However, there is an acute lack of workload generation, benchmarking, and testing frameworks that allow for comparisons between these systems or evaluate optimizations within these systems. Prior work encompassing dataframe systems either lacks a comprehensive evaluation or is evaluated against handwritten, static queries run against specific datasets [9, 14, 18, 23]. Existing systems for relational benchmarking (such as TPC [4]), or for NoSQL/key-value stores (such as YCSB [3]) do not adequately represent the workflows typically executed within these dataframe environments. DataFrame systems are a kind of middle-ground between relational systems and raw key-value stores; they do not require strict schema definitions or DDL steps like the former but are not entirely schema-less and support a vastly broader range of queries and transformations, unlike the latter.

Correctness guarantees and performance bottlenecks are harder to pinpoint with dataframe-based systems because of the diversity in query language and semantics, supported data types, serialization formats, memory layouts and execution engines. For example, the `modin` project advertises performance advantages over `pandas` by parallelizing common dataframe operations over multiple cores, ideally offering speedups of Nx where N is the number of cores available [14]. However, during the time of writing, there were ~ 45 open performance issues on the `modin` GitHub page [15], with users providing empirical evidence of `modin`'s performance being worse than `pandas` for the same operation. A more robust workload generation suite can supplement regression testing practices to catch performance and correctness bugs during the development lifecycle. Thus, there is a need for a framework that allows for synthetic workload generation, allowing for reproducibility, scalability, and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DBTest'22, June 17, 2022, Philadelphia, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9353-9/22/06…$15.00
https://doi.org/10.1145/3531348.3532178

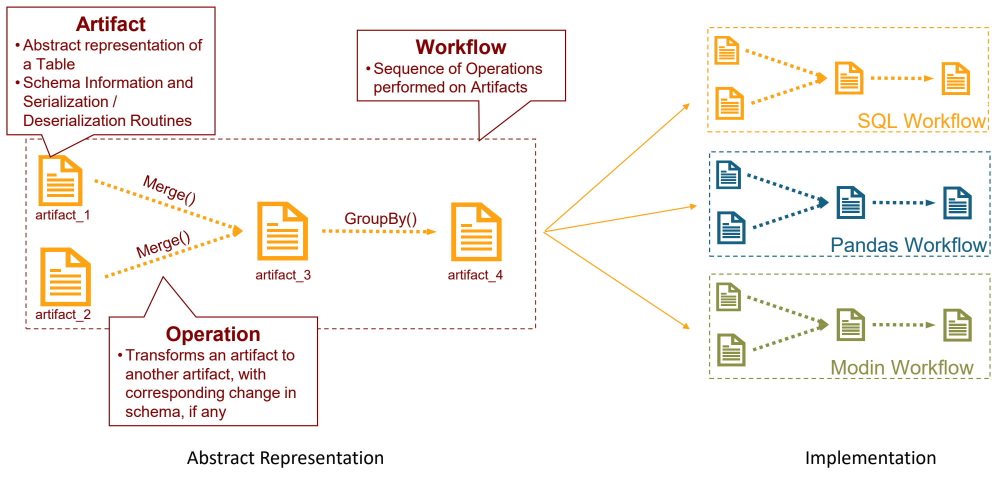

Artifact
• Abstract representation of a Table
• Schema Information and Serialization / Deserialization Routines
Workflow
• Sequence of Operations performed on Artifacts
artifact_1
artifact_2
artifact_3
artifact_4
Merge()
Merge()
GroupBy()
Operation
• Transforms an artifact to another artifact, with corresponding change in schema, if any
SQL Workflow
Pandas Workflow
Modin Workflow
Abstract Representation
Implementation

Figure 1: The design of FuzzyData.

stress testing of dataframe systems while using some of the best design practices and patterns from earlier benchmark and workload generator suites designed for relational [4] and NoSQL systems [3].

In this paper, we present FuzzyData[1]. This system allows for data analysis workflows to be manually specified or randomly generated, scaled, and replayed on various dataframe system clients, allowing for "fuzzy" testing and direct performance comparisons between dataframe systems. Our contributions are as follows:

- An implementation-agnostic data model to represent data artifacts, operations, and workflows that can be used to express common data workflow operations in JSON (a human-parsable format).
- A module that generates realistic and diverse tabular data artifacts and workflows consisting of typical data transformations.
- Plug-in clients for FuzzyData that enable workflows to be generated and replayed on `SQLite`, `pandas`, `modin-dask`, and `modin-ray`. We show how, with a few lines of code, FuzzyData can be extended to more data operations or clients by writing simple plug-ins.
- We demonstrate the use of FuzzyData in typical testing/workload generation and replay scenarios and show the types of insights that can be derived from its use. FuzzyData was used to find one correctness bug (Section 4.1) which was reported to the `modin` developers, as well as performance issues in `modin`.

2 RELATED WORK

Benchmarking and database workload generation systems have a long and storied history in academia and industry [8].

The TPC benchmark suite [4] is the most popular benchmark suite for database systems and encompasses transactional processing, data warehousing, data integration, big-data systems, and others. Vogelsgesang et al. [22] have empirically shown that typical data visualization workloads originating from BI application dashboards such as Tableau consist of complex nested sub-queries, which are very different from typical TPC workloads.

The rise in popularity of the Dataframe-based data analysis workflows has resulted in a large number of systems that implement the API while making improvements to the memory model [6, 21], leverage parallel and distributed computing [11, 13, 17], heterogeneous hardware [19] and usability improvements [12]. However, the lack of a standardized workload or performance benchmark for dataframe systems results in primarily anecdotal reports of performance improvements. One prominent example is the H20.ai benchmark website [9], which compares join and groupby query performance against dataframes of various sizes (0.5, 5, and 50GB at the time of writing); the website is designed to run periodically against the latest versions of popular "database-like" systems in an automated fashion. Among published work, the AFrame system [18] used a version of the scalable Wisconsin benchmark [8], consisting of mostly integer columns and a few string patterns with varying cardinality to show performance improvements. Sanzu [23] is a big-data benchmark suite designed to test data science environments with a wide variety of operations such as wrangling and machine learning but consists of static queries and datasets. Similarly, Böhm et al. [2] conducts a deep dive into dask's runtime and scheduler system to find performance bottlenecks; the benchmark suite is again a set of static tasks/operations performed against static datasets.

YCSB [3], is a popular NoSQL benchmark system that is aimed at cloud-serving systems that support a limited set of queries (insert/update/read/scan), with the option of scaling and mixing query types in a workload. YCSB's popularity can be attributed to its extensible client-server model – a new key-value store or database client can be written using the YCSB API and run existing workload sets to compare against other YCSB clients. We have sought to emulate this design pattern with FuzzyData.

[1]Source Code is available at https://github.com/suhailrehman/fuzzydata

3 FUZZYDATA

This section discusses the abstract design of the FuzzyData system and its implementation, along with the random table/workflow generators and the three clients we have implemented using the FuzzyData abstract model.

3.1 Data Model

The abstract model that we have created allows us to specify *artifacts*, *operations*, and *workflows* in an implementation agnostic manner:

3.1.1 Artifact. An *artifact* is an abstract representation of a dataframe in FuzzyData. The data model representing an artifact stores the metadata about an artifact, such as:

- Schema related information: A mapping of column label → column type
- Artifact representation: Information about the artifact represention in memory or as a table/view in a database
- Serialization routines: File system paths to store serialized versions of artifacts, as well as function pointers to serialization routines necessary to load/store the artifact from disk

The abstract model for an artifact gives FuzzyData the flexibility to support various data access and manipulation APIs. For example, a pandas implementation of the artifact class would typically contain the dataframe label, and a filesystem path to (de)-serialize the dataframe artifact from/to disk. On the other hand, a SQL implementation of the artifact class will have much of the same information as the pandas' implementation with additional fields such as the database table/view names, schema, and the specific SQL query needed to retrieve a view of the artifact from a database.

3.1.2 Operation. An *operation* is the abstract representation of one or more transformation(s) that takes one or more artifact(s) as input and transforms them into a new artifact. The *operation* model consists of metadata such as the source artifact labels, a list of transformations and their arguments, and the label to be assigned to the new destination artifact created as a result of the operation. The *operation* interface consists of abstract specifications of each of the transformations with their arguments and how each of the transformations change the schema of the artifact. Note that the actual query or code required to execute the transformations for a particular operation is defined in the client implementation and left out the abstract model. Table 1 has a listing of all the transformations that are currently implemented in FuzzyData. The operator abstraction is designed to be extensible, allowing for additional transformations or UDFs to be added to FuzzyData to support even more diverse workload types in the future. Transformations can be defined and implemented for data processing steps such as one-hot-encoding or normalization.

3.1.3 Workflow. Formally, a *workflow* is a directed-acyclic-graph (DAG) $\mathcal{W} = (V, E)$, where the set of vertices V is the set of artifacts and the set of edges E are the operations that are used to transform an artifact $v_1 \in V$ to another artifact $v_2 \in V$. Artifacts that have no incoming edges are *source artifacts*, which are either loaded from disk, or randomly generated. Note that the set of edges E in FuzzyData is *ordered*, and is the sequence of operations that is implicit when encoding an existing workflow (Section 3.2), or is the order of operations that is randomly generated by FuzzyData (Section 3.4).

In FuzzyData, the *workflow* interface encapsulates information about the workflow, sequence of operations, and the final workflow directed acyclic graph, with enough information for FuzzyData to load and replay specific workflows on different clients (Section 3.6). As a workflow is replayed, the wall-clock time taken to load artifacts from the disk and replay the individual operations are recorded for analysis and visualization (Section 3.5).

3.2 Implementation

FuzzyData is implemented in Python, with `pandas` [20] used as the internal dataframe representation used to represent artifacts during generation (Section 3.3) and `networkx` [10] used to represent the workflow graph. Users of FuzzyData have the option to specify a workflow or have FuzzyData randomly generate a workflow (Section 3.3) with generation parameters listed in Table 3.

FuzzyData supports workflow specifications to be provided in a JSON file. The workflow specification includes a name for the workflow and a list of operations. Each operation consists of a list of source artifacts, the operation type, and the arguments for the operation. When a workflow specification and the source artifacts are loaded into a FuzzyData client, the following actions are performed: (a) The operation list is loaded, (b) the operations are performed in the order specified in the file, and (c) any artifact that is currently not in memory and is needed for the next operation will be deserialized from disk.

3.3 Generating Random Artifacts

FuzzyData provides a table generation system that leverages the Python faker library [5]. The faker integration allows FuzzyData to generate diverse tables with a wide range of columns and data, simulating real-world datasets with realistic data values. A user can supply the number of rows (r) and columns (c) to be generated, and FuzzyData by default randomly chooses columns types to generate from faker's portfolio and generates a mix of string and numeric columns with various levels of cardinality. These column types are labeled as one or more of the following types: *numeric, string, groupable* and *joinable*; these labels are essential when enumerating the space of possible operations that can be performed on a given artifact (enumerated in Table 1. Table 2 shows an example table generated with parameters ($r = 10, c = 4$) with a mix of column types and labels. This table can now be used to generate additional artifacts based on the rules described below in Section 3.4.

3.4 Generating and Replaying Random Workflows

FuzzyData also supports the generation of entirely random workflows, including randomly generated source artifacts (using the generator described in Section 3.3) and random operations. Users have the option of supplying the following parameters (Table 3):

Users can specify the total number of artifacts to be generated (n) and the number of rows (r) and columns (c) of the base artifact, and a few schema options, if any. The schema options in the generator

Transformation	**Description**	**Generation Constraints**	`pandas`	`SQLite`	`modin`
`load`	Load (de-serialize) an artifact from a filesystem location	–	✓	✓	✓
`select`	Selects rows based on a filter condition	`numeric` ≥ 1	✓	✓	✓
`apply`	Create a new derived column as a scalar function applied to an existing numeric column	`numeric` > 1	✓	✓	✓
`project`	Project a set of columns	–	✓	✓	✓
`sample`	Randomly select rows from the artifact	–	✓	✓	✓
`join`	Inner Join with another artifact based on a key column	`joinable` ≥ 1	✓	✓	✓
`pivot`	Pivot the artifact by index, column and values	`groupable` ≥ 2; `numeric` ≥ 1	✓	–	✓
`groupby`	Groupby set of `group_columns` and apply an aggregate `function` on another set of `agg_columns`	`groupable` ≥ 1	✓	✓	✓
`fill`	Replace an `old` value in a column with another value	–	✓	–	✓
`materialize`	Execute stacked transformations to produce a new artifact	–	✓	✓	✓
`serialize`	Dump the contents of an artifact to disk	–	✓	✓	✓

Table 1: Transformation Implementation Matrix. The *Generation Constraints* column lists the minimum number of columns of each type required to generate each transformation using the random workflow generator.

index	iso8601	cryptocurrency_code	pyint	rn
0	2004-09-21T09:46:38	NEO	1453	10
1	2016-04-07T21:19:57	BCN	877	7
2	1973-08-09T20:35:50	USDT	8198	8
3	1985-08-24T17:07:41	EOS	7492	10
4	1979-06-16T21:01:12	NEM	157	6
5	2020-12-30T03:19:01	IOTA	5439	12
6	1995-05-03T04:56:00	BCH	2348	13
7	1972-09-02T20:03:53	XRP	1244	13
8	1990-12-27T10:33:05	ETC	8354	11
9	2017-07-03T20:19:32	WAVES	9717	11

Table 2: An example artifact generated using FuzzyData. Column labels indicate the faker provider used to generate the values for the column, except for `rn`, which is shorthand for `random_number`. In this table, `pyint` and `rn` are *numeric* columns, while `cryptocurrency_code` is a *groupable, joinable* and *string* column.

Parameter	**Description**
n	Number of Artifacts
r	Base Artifact Number of Rows
c	Base Artifact Number of Columns
b	Workflow Branching Factor
T	Set of Allowed Transformations
m	Materialization Rate

Table 3: Parameters for generating synthetic workflows.

can provide specific column types to be generated or an exclusion list of the types of columns to avoid. If no schema options are provided, the generator will try to evenly distribute the types of columns in order to maximize the available operations that can be performed (Table 1). Once the base artifact is generated, subsequent artifacts are generated as follows:

(1) A random artifact is first selected from the set of artifacts already generated in the workflow. The *workflow branch factor* parameter (b) can be used to control the structure of the final workflow graph by biasing the selection probability towards artifacts that were more recently generated. Formally, given we have a list of n' artifacts that have been generated so far, FuzzyData selects the next artifact from the list with index i to be modified with probability (Equation 1)

$$P[i] = \left(\frac{b}{e^{bn'} - 1}\right) e^{bi} \tag{1}$$

Thus, with $b = 1.0$, the probability is always skewed towards the newest artifact that is generated, resulting in workflow that is more linear, and with $b = 0.01$, there is a uniform probability, resulting in a more branched workflow (such as the example workflow generated in Figure 2).

(2) Once an artifact has been selected as the source artifact for an operation, FuzzyData generates a set of possible transformations that can be performed on the artifact, subject to the set of allowed transformations T. The source artifact schema map is inspected, and the number of columns of each type is enumerated. It then follows the rules listed in Table 1 and generates a set of transformations with randomized arguments for each transformation.

(3) From the set of possible transformations that can now be performed on a source artifact, a random option is selected and added to the operation chain. The expected schema map that results from the transformation is updated, ensuring any future stacked transformations have accurate schema representation, even if the operation is not materialized.

(4) Steps 2 and 3 are repeated until we have m transformations in the current operation chain, i.e. we have reached the materialization rate.

(5) The operation chain is materialized (executed) to generate the next artifact. This means that FuzzyData will use an attached client (Section 3.6) to execute the chain of operations and generate the resulting artifact. If the generator selects a merge operation to be performed, an additional random artifact that contains the *joinable* column (from the

merge arguments) is generated and added to the workflow to simulate an inner PK-FK join.

(6) The steps above are repeated until we have generated n artifacts and there are no more artifacts that remain to be generated.

(7) Once the workflow is generated, it can be written to disk, which serializes all generated artifacts to disk, writes the workflow graph, and generates a JSON specification of the workflow, which can be loaded and replayed by FuzzyData clients in the future.

Figure 2 is an example of the DAG associated with a workflow generated in FuzzyData. While step 2 in our workflow generator currently implements a simple, randomized, rule-based approach for generating operations on artifacts, we foresee the ability to use intelligent, ML-based approaches to automatically generate meaningful operations given the source artifact, similar to AutoSuggest [24], implemented via the plug-in architecture of FuzzyData.

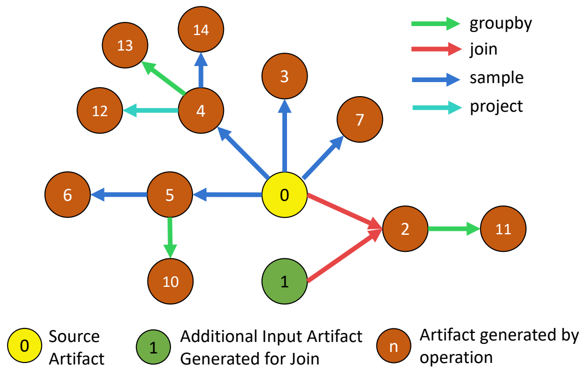

groupby
join
sample
project
0
1
2
3
4
5
6
7
10
11
12
13
14
Source Artifact
Additional Input Artifact Generated for Join
n
Artifact generated by operation

Figure 2: DAG of a randomly generated workflow using the generator described in Section 3.4. The generation parameters used were $(n = 15, r = 1000, c = 20, b = 0.01, m = 1)$**.**

3.5 Instrumentation

FuzzyData includes timing hooks implemented into the *workflow* interface for artifact generation, loading, and operations. These hooks can be used to collect runtime performance information for the execution of the workflow. This feature makes FuzzyData valuable for system testing and performance analysis and evaluation of potential bottlenecks in the systems being evaluated.

3.6 Clients Implemented

Implementing a FuzzyData client requires implementing each of the abstract interfaces described in Section 3. The *artifact* interface requires implementing load/store routines and a hook to the random artifact generation function. The *operation* interface requires implementing each of the operations listed in Table 1, using the specific syntax of the client's query language or DSL. The *workflow* interface provides space for setting up parameters used by all the artifacts in the workflow, like filesystem paths or database/execution engine parameters. Thus, with a few lines of code, a client can be implemented in FuzzyData that can generate and replay workflows. Three clients have been implemented in FuzzyData for this paper:

`pandas`: The `pandas` client is a dataframe-based implementation of the three abstract interfaces described in Section 3. The *operation* implementation generates dataframe transformation code as a string of chained dataframe function calls. To materialize an artifact, the string containing all the dataframe transformations is evaluated and run against the source dataframe artifact. Appendix B lists our client implementation code in its entirety.

`modin`: The `modin` client is an extension of the `pandas`' client, in which all `pandas` operations are simply routed through the `modin` library [14]. The client provides the user an option of specifying either a `dask` or `ray` execution engine, along with initialization parameters such as number of workers, which can distribute the dataframe operations on parallel hardware.

`SQLite`: The `SQLite` client creates a file-based embedded database on disk and uses the generator described in Section 3.3 to generate base table artifacts. The base table artifacts are then operated upon by SQL queries constructed for each operation to generate other artifacts as views in the database. We use nested sub-queries to chain all of the transformations into an operation. All table views are serialized to disk as CSV files at the end of the workflow. A notable exclusion from the `SQLite` client is the `pivot` operation since generic pivots in a generic SQL dialect are quite complex to generate. The client initialization parameters include an SQL connection string, so this client can be used with other SQL databases as well. In case of a SQL dialect mismatch for other database systems, this client could be extended to re-implement any of the incompatible operations.

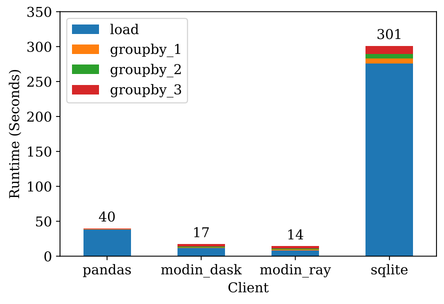
4 USE CASES

In this section, we demonstrate the various uses of FuzzyData. We first recreate a simple workflow using a publicly available Jupyter notebook. We then generate a randomized workflow, scale the workflow size and show the runtime performance on the three clients. All of our experiments were run on a server running Ubuntu 18.04, with an Intel Xeon Silver 4416 CPU, 196 GB RAM . All of our code was executed using the Python 3.8.5 interpreter with `pandas` v1.4.0, `SQLite` v3.33.0, `modin` v0.13.2, `dask` v2022.2.0, and `ray` v1.10.0. The `dask` engine was configured using default settings, which resulted in 8 processes and 48 threads being spawned for each bechmarking session. `ray` was also configured similarly, resulting in upto 48 worker threads being spawned in each session.

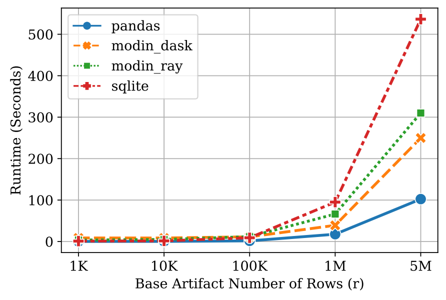
4.1 Fuzzy Testing Suite for Dataframe Systems

Any client implemented in the FuzzyData library can use the built-in test suite, generating many workflow test cases with variable number of artifacts, rows, columns and operations[2]. These tests can be used to check API and result equivalences and corner cases for operations run using diverse column types. Using FuzzyData's test suite, we uncovered an API corner case in `modin`, wherein a

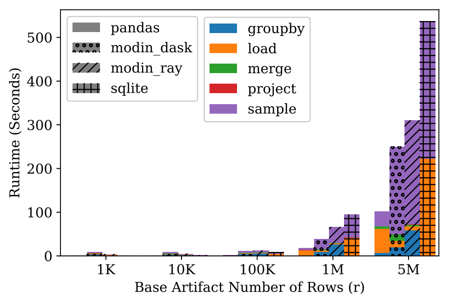
[2]https://github.com/suhailrehman/fuzzydata/tree/main/tests



(a) Real-World Workflow Replay



(b) Scaling Experiment -Total Runtime



(c) Scaling Experiment - Breakdown of runtime.

Figure 3: Results of the Workload Replay and Scaling Experiments

dataframe generated in memory fails to execute consecutive group-bys, and reported the issue to the `modin` developers[3]. The issue stems from lazy metadata propagation in `modin`.

4.2 Encoding and Replaying an Existing Workflow

In this experiment, we encode an existing workflow from the `modin` examples page [16]. The encoded JSON file for this workflow is in Appendix A. The primary artifact that is loaded into this workflow is a 1.8GB CSV file, following which we execute three different group-by operations on the same artifact. The resulting execution timeline is depicted in Figure 3a. We can see that the `modin-ray` client completes the workflow the fastest at approximately 2.8x faster than `pandas` and 21.5x faster than `SQLite`, with the runtime being dominated by the CSV loading process.

4.3 Scaling and Replaying a Generated Workflow

Figure 2 represents a workflow generated by FuzzyData with parameters $(n = 15, r = 1000, c = 20, b = 0.01, m = 1)$, with all operations permitted except `pivots`. The workflow was loaded and replayed in the three FuzzyData clients with the results shown in Figures 3b.

For the workflow generated in Section 3.4, we scaled the base artifact up from 1000 rows up to 5 million and re-ran the workload on all the clients. Figure 3b shows the results of our scaling experiment. This specific example shows that `pandas` outperforms all the other clients, even at 5M rows (2.1 GB). Figure 3c illustrates the runtime breakdown grouped by operation. Despite the improvement in loading times, the total time to draw the six random samples from the distributed dataframes is much slower than the corresponding operation in `pandas`, slowing down the total runtime in `modin` compared to `pandas`, indicating a potential avenue for improvement in `modin`.

5 CONCLUSIONS AND FUTURE WORK

We have shown FuzzyData to be a useful workflow generation system that can be used to generate workflows and test/evaluate dataframe-style systems. Due to the pluggable nature of our implementation, many extensions can be considered for future work:

- Further enhancement of the randomized table generator, allowing users to express inter-column functional dependencies and expected cardinalities for columns.
- The rule-based operation/workflow generator can be extended to use learned features to automatically generate even more realistic operations and arguments, based on artifact features, such as those described in [24].
- Integrate additional serialization formats such as `parquet` to enable comparisons across serialization formats.
- Additional clients can be implemented in FuzzyData to provide even more points of comparison.

[3]https://github.com/modin-project/modin/issues/4287

REFERENCES

[1] Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, and Matei Zaharia. 2015. Scaling spark in the real world: performance and usability. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1840–1843. https://doi.org/10.14778/2824032.2824080

[2] Stanislav Böhm and Jakub Beránek. 2020. Runtime vs Scheduler: Analyzing Dask's Overheads. In *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. 1–8. https://doi.org/10.1109/WORKS51914.2020.00006

[3] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*. ACM Press, Indianapolis, Indiana, USA, 143. https://doi.org/10.1145/1807128.1807152

[4] TPC Council. 2022. TPC-Homepage. https://tpc.org/default5.asp

[5] Daniele Faraglia and Other Contributors. 2022. Faker. https://github.com/joke2k/faker original-date: 2012-11-12T23:00:09Z.

[6] Apache Software Foundation. 2020. Apache Arrow. https://arrow.apache.org/

[7] The R Foundation. 2022. R: The R Project for Statistical Computing. https://www.r-project.org/

[8] Jim Gray (Ed.). 1994. *The Benchmark handbook: for database and transaction processing systems* (2. ed., 2. [print.] ed.). Morgan Kaufmann, San Francisco, Calif.

[9] H20.ai. 2022. Database-like ops benchmark. https://h2oai.github.io/db-benchmark/

[10] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using networkx.* Technical Report LA-UR-08-05495; LA-UR-08-5495. Los Alamos National Lab. (LANL), Los Alamos, NM (United States). https://www.osti.gov/biblio/960616

[11] Andreas Kunft, Lukas Stadler, Daniele Bonetta, Cosmin Basca, Jens Meiners, Sebastian Breß, Tilmann Rabl, Juan Fumero, and Volker Markl. 2018. ScootR: Scaling R Dataframes on Dataflow Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 288–300. https://doi.org/10.1145/3267809.3267813

[12] Doris Jung-Lin Lee, Dixin Tang, Kunal Agarwal, Thyne Boonmark, Caitlyn Chen, Jake Kang, Ujjaini Mukhopadhyay, Jerry Song, Micah Yong, Marti A. Hearst, and Aditya G. Parameswaran. 2021. Lux: always-on visualization recommendations

for exploratory dataframe workflows. *Proceedings of the VLDB Endowment* 15, 3 (Nov. 2021), 727–738. https://doi.org/10.14778/3494124.3494151

[13] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[14] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards scalable dataframe systems. *Proceedings of the VLDB Endowment* 13, 12 (Aug. 2020), 2033–2046. https://doi.org/10.14778/3407790.3407807

[15] Modin Project. 2022. Issues · modin-project/modin. https://github.com/modin-project/modin

[16] Modin Project. 2022. Modin NYC Taxi Example Notebook. https://github.com/modin-project/modin/blob/dd9beee3a599d3a91036cbaeef8b8499ba9cc4c1/examples/jupyter/NYC_Taxi.ipynb original-date: 2018-06-21T21:35:05Z.

[17] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*. Austin, TX, 130–136.

[18] Phanwadee Sinthong and Michael J. Carey. 2019. AFrame: Extending DataFrames for Large-Scale Modern Data Analysis. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 359–371. https://doi.org/10.1109/BigData47090.2019.9006303

[19] RAPIDS Development Team. 2018. RAPIDS: Collection of Libraries for End to End GPU Data Science. https://rapids.ai

[20] The pandas development team. 2020. pandas-dev/pandas: Pandas. https://doi.org/10.5281/zenodo.3509134

[21] Ritche Vink. 2021. Polars. https://github.com/pola-rs/polars original-date: 2020-05-13T19:45:33Z.

[22] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the Workshop on Testing Database Systems (DBTest'18)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3209950.3209952

[23] Alex Watson, Deepigha Shree Vittal Babu, and Suprio Ray. 2017. Sanzu: A data science benchmark. In *2017 IEEE International Conference on Big Data (Big Data)*. 263–272. https://doi.org/10.1109/BigData.2017.8257934

[24] Cong Yan and Yeye He. 2020. Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, Portland, OR, USA, 1539–1554. https://doi.org/10.1145/3318464.3389738

A WORKFLOW SPEC USED IN SECTION 4.2

The following JSON file is a manually encoded list of operations derived from the workflow example in [16]. This JSON input file and source artifact can be used as input to replay the workflow on all FuzzyData clients. The runtime result of this workflow on all clients is in Section 4.2.

```json
{
  "name": "nyc-cab",
  "operation_list": [{
    "sources": [
      "yellow_tripdata_2015-01"
    ],
    "new_label": "gb_vendor_id_amount_sum",
    "operation_list": [{
      "op": "groupby",
      "args": {
        "group_columns": [
          "VendorID"
        ],
        "agg_columns": [
          "total_amount"
        ],
        "agg_function": "sum"
    }}]},
    {
      "sources": [
        "yellow_tripdata_2015-01"
      ],
      "new_label": "gb_pcount_amount_mean",
      "operation_list": [{
      "op": "groupby",
      "args": {
        "group_columns": [
          "passenger_count"
        ],
        "agg_columns": [
          "total_amount"
        ],
        "agg_function": "mean"
      }}]
    },
    {
      "sources": [
        "yellow_tripdata_2015-01"
      ],
      "new_label": "gb_pcount_vendor_total_mean",
      "operation_list": [{
      "op": "groupby",
      "args": {
        "group_columns": [
          "passenger_count",
          "VendorID"
        ],
        "agg_columns": [
          "total_amount"
        ],
        "agg_function": "mean"
      }}]
    }
  ]
}
```

B PANDAS CLIENT IMPLEMENTATION

We show a compact client implementation of FuzzyData on pandas below, which implements the interfaces defined in Section 3, along with the operations listed in Section 1.

```python
import logging
from typing import List

import pandas

from fuzzydata.core.artifact import Artifact
from fuzzydata.core.generator import generate_table
from fuzzydata.core.operation import Operation, T
from fuzzydata.core.workflow import Workflow

logger = logging.getLogger(__name__)


class DataFrameArtifact(Artifact):

    def __init__(self, *args, **kwargs):
        self.pd = kwargs.pop("pd", pandas)
        from_df = kwargs.pop("from_df", None)
        super(DataFrameArtifact, self).__init__(*args, **kwargs)
        self._deserialization_function = {
            'csv': self.pd.read_csv
        }
        self._serialization_function = {
            'csv': 'to_csv'
        }

        self.operation_class = DataFrameOperation
        self.table = None
        self.in_memory = False

        if from_df is not None:
            self.from_df(from_df)

    def generate(self, num_rows, schema):
        self.table = generate_table(num_rows, column_dict=schema, pd=self.pd)
        self.schema_map = schema
        self.in_memory = True

    def from_df(self, df):
        self.table = self.pd.DataFrame(df)
        self.in_memory = True

    def deserialize(self, filename=None):
        if not filename:
            filename = self.filename

        self.table = self._deserialization_function[self.file_format](filename)
        self.in_memory = True

    def serialize(self, filename=None):
        if not filename:
            filename = self.filename

        if self.in_memory:
            serialization_method = getattr(self.table, self.
        _serialization_function[self.file_format])
            serialization_method(filename)

    def destroy(self):
        del self.table

    def to_df(self) -> pandas.DataFrame:
        return self.table

    def __len__(self):
        if self.in_memory:
            return len(self.table.index)
```

```


class DataFrameOperation(Operation['DataFrameArtifact']):
    def __init__(self, *args, **kwargs):
        self.artifact_class = kwargs.pop('artifact_class', DataFrameArtifact)
        super(DataFrameOperation, self).__init__(*args, **kwargs)
        self.code = 'self.sources[0].table' # Starting point for chained code generation.

    def apply(self, numeric_col: str, a: float, b: float) -> DataFrameArtifact:
        super(DataFrameOperation, self).apply(numeric_col, a, b)
        new_col_name = f"{numeric_col}__{int(a)}x_{int(b)}"
        return f'.assign({new_col_name} = lambda x: x.{numeric_col}*{a}+{b})'

    def sample(self, frac: float) -> DataFrameArtifact:
        super(DataFrameOperation, self).sample(frac)
        return f'.sample(frac={frac})'

    def groupby(self, group_columns: List[str], agg_columns: List[str], agg_function: str) -> T:
        super(DataFrameOperation, self).groupby(group_columns, agg_columns, agg_function)
        return f'[{group_columns+agg_columns}].groupby({group_columns}).{agg_function}().reset_index()'

    def project(self, output_cols: List[str]) -> T:
        super(DataFrameOperation, self).project(output_cols)
        return f'[{output_cols}]'

    def select(self, condition: str) -> T:
        super(DataFrameOperation, self).select(condition)
        return f'.query("{condition}")'

    def merge(self, key_col: List[str]) -> T:
        super(DataFrameOperation, self).merge(key_col)
        return f'.merge(self.sources[1].table, on="{key_col}")'

    def pivot(self, index_cols: List[str], columns: List[str], value_col: List[str], agg_func: str) -> T:
        super(DataFrameOperation, self).pivot(index_cols, columns, value_col, agg_func)
        return f'.pivot_table(index={index_cols}, columns={columns},values={value_col},aggfunc={agg_func})'

    def fill(self, col_name: str, old_value, new_value):
        super(DataFrameOperation, self).fill(col_name, old_value, new_value)
        return f'.replace({{ "{col_name}": {old_value} }}, {new_value})'

    def chain_operation(self, op, args):
        self.code += getattr(self, op)(**args)
        super(DataFrameOperation, self).chain_operation(op, args)

    def materialize(self, new_label):
        new_df = eval(self.code)
        super(DataFrameOperation, self).materialize(new_label)
        return self.artifact_class(label=self.new_label,
                                   from_df=new_df,
                                   schema_map=self.current_schema_map)


class DataFrameWorkflow(Workflow):
    def __init__(self, *args, **kwargs):
        super(DataFrameWorkflow, self).__init__(*args, **kwargs)
        self.artifact_class = DataFrameArtifact
        self.operator_class = DataFrameOperation

    def initialize_new_artifact(self, label=None, filename=None, schema_map=None):
        return DataFrameArtifact(label, filename=filename, schema_map=schema_map)
```